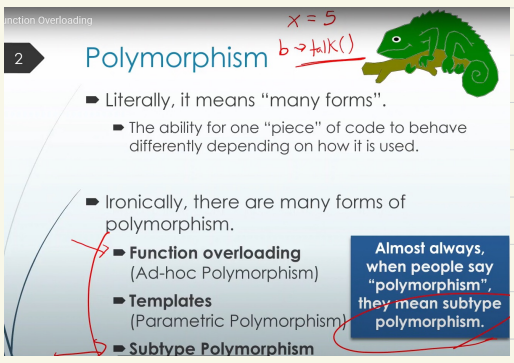
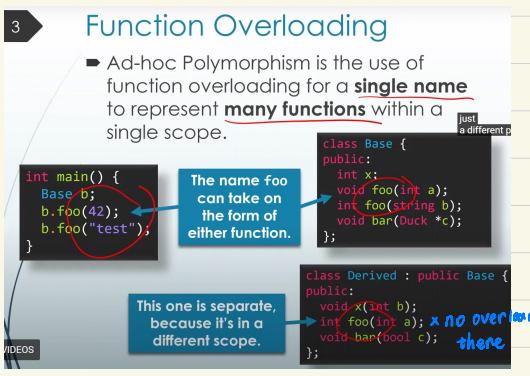
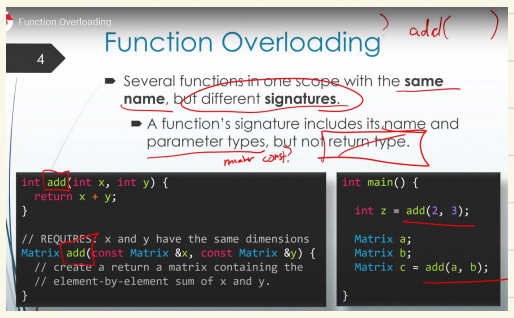
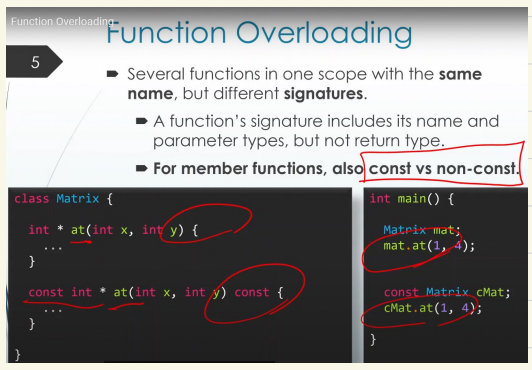
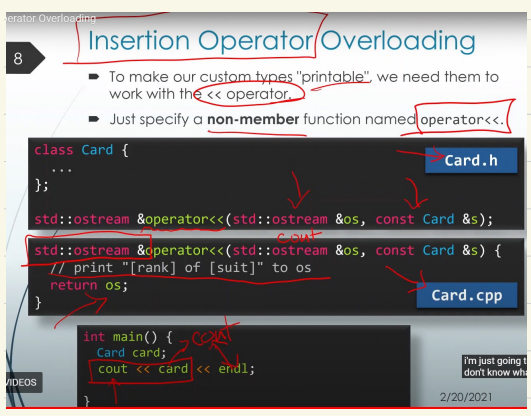
Video 1: introduction to polymorphism and function overloading







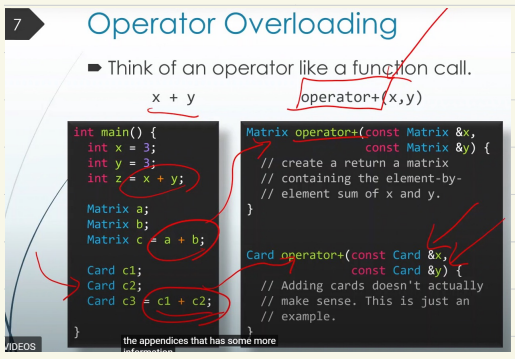




Video 2: operator overloading

Operator Overloading

- A philosophy of C++ is that user-defined types should have just as much support as built-in types.
- We can use operators (e.g. +, -, [], etc.) with our own types too!
- To do this, we just have to tell C++ what we want each operator to do.
  - The mechanism for this is **operator overloading**.



Exercise

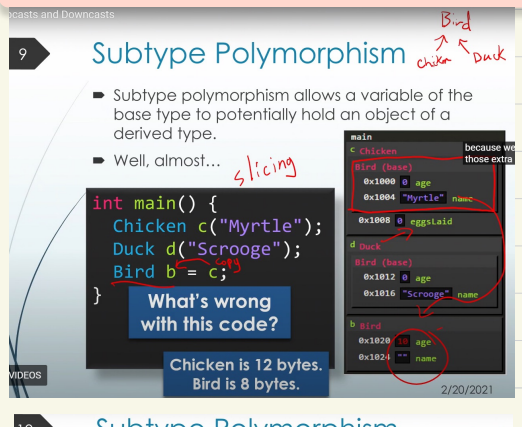
Let's upgrade `Pixel` from project 2 to a `class` and add some overloaded operators:

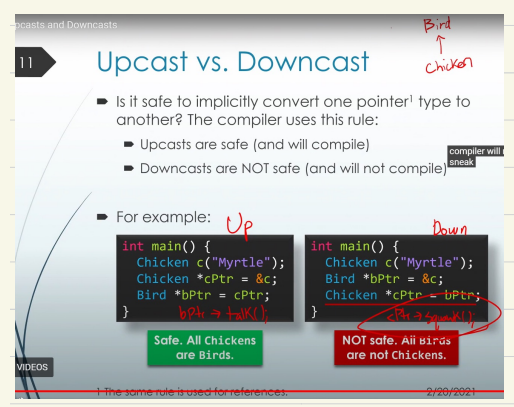
- An overloaded `<<` operator that prints a pixel in the format `rgb(R,G,B)`
- An overloaded `-` operator that computes the squared difference of two pixels

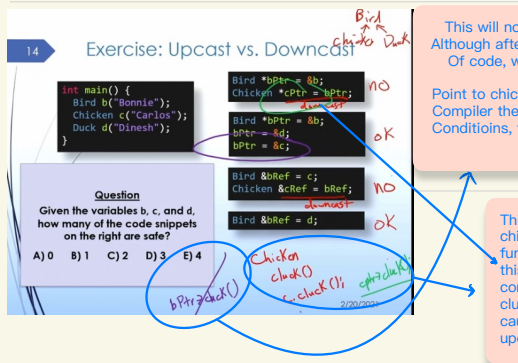
Note that both operators are implemented as non-member functions. Starter code (and a few more instructions in the comments) is available on Lobster (L10.1_Pixel). Implement the operators so that the code in main works correctly.

```
#include <iostream>
using namespace std;

class Pixel {
public:
  const int r;
  const int g;
  const int b;

  Pixel(int r, int g, int b)
    : r(r), g(g), b(b) { }

};

int squared_difference(const Pixel &p1, const Pixel &p2);

// TASK 1: Add an overloaded operator- that
// returns the squared difference between two
// pixels (you can just call squared_difference
// in your implementation)

int operator-(const Pixel &p1, const Pixel &p2) {
  return squared_difference(p1, p2);
}

// TASK 2: Add an overloaded operator<< that
// prints out the pixel in this format:
//   rgb({R},{G},{B})
ostream &operator<<(ostream &os, const Pixel &p) {
  os << "rgb(" << p.r << ", " << p.g
     << ", " << p.b << ")";
  return os;
}

int main() {
  Pixel p1(174, 129, 255);
  Pixel p2(166, 226, 46);

  cout << "p1: " << p1 << endl; // p1: rgb(174,129,255)
  cout << "p2: " << p2 << endl; // p2: rgb(166,226,46)

  cout << "sq diff: " << p2 - p1 << endl; // sq diff: 531
}

// From processing.cpp in P2 starter code
int squared_difference(const Pixel &p1, const Pixel &p2) {
  int dr = p2.r - p1.r;
  int dg = p2.g - p1.g;
  int db = p2.b - p1.b;
  // Divide by 100 is to avoid possible overflows
  // later on in the algorithm.
  return (dr*dr + dg*dg + db*db) / 100;
}
```

Video 3: subtype polymorphism(upcasts and downcasts)



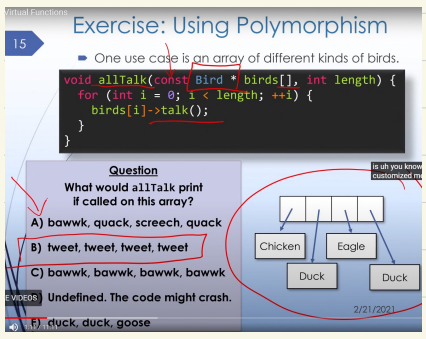


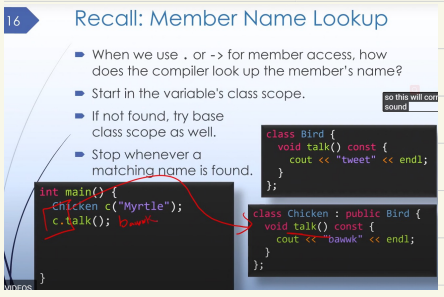


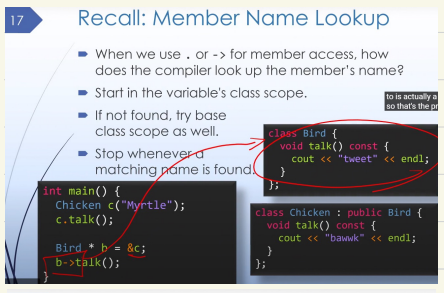
This will not compile because: Although after running other parts Of code, we know the bptr is actually Point to chicken, however, for the Compiler they don't know if other Conditioins, the bref may point to Duck.

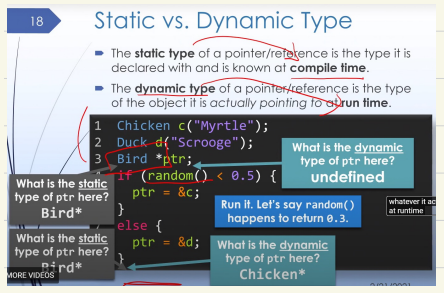
This is definitely ok since "c" is chicken and cluck() is a spectral function for chicken.However, if this also be allowed by the compiler, there are no cluck()function in bird and will cause serious error. So only the upcases are safe.

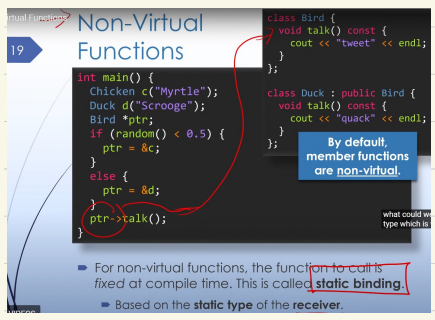
Video 4: virtual functions

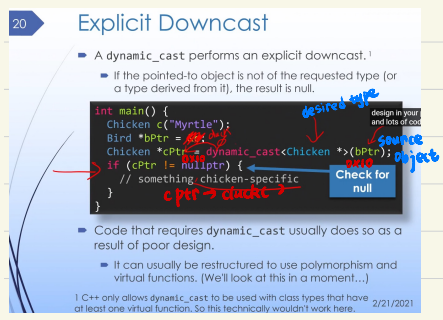




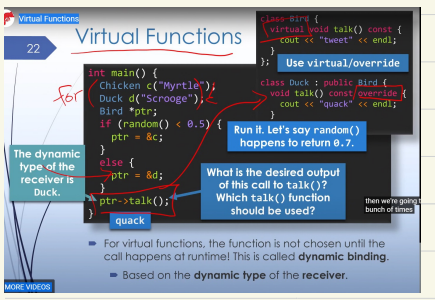


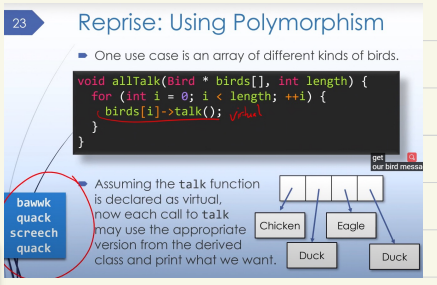






What we did is called manual Dynamic binding. By checking the dynamic type at run time to decide which function be called.
However, we can let the compiler do the checking work for us ——when we declare a function which we want to have dynamic binding all "virtual" and "override"



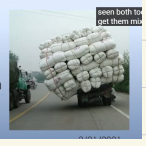


Video 5: overriding

29 Overriding vs. Overloading

- Overriding
  - Allowing a subclass to redefine the behavior for one of its inherited methods.

- Overloading
  - A single name can refer to many different functions, depending on their parameter types.



30 The override Keyword

- The override keyword tells the compiler to sanity check that the overriding function **signature actually matches something in the base class.**

```
class Bird {
  virtual void talk() const { cout << "tweet" << endl; }
};
class Duck : public Bird {
  void tak() const { cout << "quack" << endl; }
};
class Duck : public Bird {
  void tak() const override { cout << "quack" << endl; }
};
class Duck : public Bird {
  void talk() override { cout << "quack" << endl; }
};
```

Compiler allows, but it's probably a mistake!

Error: no tak() function in the base class.

Error: Doesn't match the const in base.

video 6: pure virtual functions and abstract classes

31 Awkward Bird

- Should we really have a class that is "just a bird"?
- There's no such thing. Real world birds all belong to a more specific kind.
- The `talk()` function is awkward. There's really no reason to pick "tweet".
- It's easy to imagine other functions for which the problem is worse.

```
class Bird {
private:
  int age;
  string name;

public:
  Bird(string name_in)
    : age(0), name(name_in) {
    cout << "Bird ctor" << endl;
  }
  ...
  virtual void talk() const {
    cout << "tweet" << endl;
  }
  virtual int getWingspan() const {
    return -1; // ???
  }
};
```

32 Pure Virtual Functions

- Instead, we can make some functions **pure virtual**.
- This means that the definition of the function is "there is no implementation".
- Bird is now an **abstract class**.
- You can't make any instances of an abstract class, but you wouldn't want to!
- Subclasses must[1] **override** the function to provide an implementation.

```
class Bird {
private:
  int age;
  string name;

public:
  Bird(string name_in)
    : age(0), name(name_in) {
    cout << "Bird ctor" << endl;
  }
  ...
  virtual void talk() const = 0;
  virtual int getWingspan()
    const = 0;
};
```

[1] Otherwise, the function remains as pure virtual and the subclass is also abstract.

34 Interfaces

- An **interface** is a class where all functions are pure virtual.

```
class Shape {
public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  virtual void scale(double s) = 0;
};
```

- Other classes (e.g. `Triangle`, `Rectangle`) that derive from `Shape` are said to "implement" the `Shape` interface.

35 Inheriting Interfaces

- We've already seen one reason to use inheritance is to save on code duplication in classes with shared functionality.
- Another, equally important use is to create hierarchies of types that all implement a base class interface.
  - Polymorphism allows functions or other code that work with the base interface to work with any object that inherits from it.
  - An example of this from the Euchre project is that your driver code only needs to worry about working with `Player*`, but it can then support any combination of simple or human players!

Video 7: the liskov substitution principle

Subtypes vs. Derived Types

- Not all derived types are subtypes!
  - Anything that inherits members is a derived type.
  - A subtype must follow the is-a relationship.
- But what does the "is-a" relationship mean, exactly?
  - Liskov Substitution Principle

Liskov Substitution Principle

- If S is a subtype of T...
  - Any property of T should also be a property of S.
  - In any code that depends on T's interface, an object of type S can be **substituted** without any undesirable effects.

Barbara Liskov, MIT

36 Example: Ducks

```
class Duck : public Bird {
  ...
  // EFFECTS:  prints out quack
  void talk() const override {
    cout << "quack" << endl;
  }
};
```

```
class ToyDuck : public Duck {
  ...
  // EFFECTS:  prints out quack if
  //           batteryLevel >= 10
  void talk() const override {
    if (batteryLevel >= 10) {
      cout << "quack" << endl;
      --batteryLevel;
    }
  }
};
```

38 Dynamic Polymorphism

- With subtype polymorphism, we can make the decision on what type to use at runtime.

```
int main() {
  string color;
  cin >> color;
  Bird *bird = Bird_factory(color, "Myrtle");
  bird->haveBirthday();
  cout << "Bird " << bird->getName()
       << " is " << bird->getAge()
       << " and says: " << bird->talk() << endl;
  delete bird;
  cin >> color;
  bird = Bird_factory(color, "Heihei");
  cout << "Bird " << bird->getName()
       << " is " << bird->getAge()
       << " and says: " << bird->talk() << endl;
  delete bird;
}
```

39 The Factory Pattern

- A factory function is a function that creates and returns objects.

```
Bird * Bird_factory(const string &color,
                    const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
  ...
}
```

If we don't use the key word new, it will creat a local variable which lifetime will dead after the function return. So we would be returning a pointer to a dead object

```
//动态多态满足条件
// 1、有继承关系
// 2、子类重写父类的虚函数

//动态多态使用
// 父类的指针或者引用 执行子类对象
```